Visual-Trace Simulation of Concurrent Finite-State Machines for Validation and Model-Checking of Complex Behaviour

Robert Coleman, Vladimir Estivill-Castro, René Hexel, and Carl Lusty

Griffith University, Nathan 4111, QLD, Australia
www.mipal.net.au

Abstract. Simulation of models that specify behaviour of software in robots, embedded systems, and safety critical systems is crucial to ensure correctness. This is particularly important in conjunction with model-driven development, which is highly prevalent due to its numerous benefits. We use vectors of finite-state machines (FSMs) as our modelling tool. Our FSMs can have their transitions labeled by expressions of a common sense logic, and they are more expressive than other modelling approaches (such as Behavior Trees, Petri nets, or plain FSMs). We interpret the models using the same round-robin scheduler which is integrated into the simulator. Execution on a platform is exactly the same as in the simulator (where sensors and actuators are masqueraded by proxies) and coincides with the generator of the Kripke structure for formal model-checking. In three ubiquitous case studies we show that our simulation discovers issues where those models were incomplete, ambiguous, or incorrect. This further illustrates that simulation and monitoring need to complement formal verification.

Keywords: simulation, testing and validation of robot software, interpretation of models, model-checking, modeling framework for robots, software platform and middleware for robotics.

1 Introduction

While software development for autonomous robots commonly starts with a modelling phase, enough confidence in the correctness of behaviour can only be obtained through simulation in conjunction with formal model-checking.

Model-driven engineering minimises the programming phase as correct, and validated models can directly run on different platforms. Model-driven development (MDD) has been shown to be fast, as it provides higher level of abstraction than traditional programming languages. Models are automatically transformed into efficient, working software. In fact, models can more succinctly represent functionality. MDD is more cost-effective, due to a shorter time to deployment, but changes to models and traceability of functionality are more direct and transparent. This also leads to increased quality. MDD is less error-prone if we can perform testing, validation, and simulation of models. Therefore, MDD leads to meaningful validation, as it is the high-level model that is validated, because automatic translation to lower levels has perfectly defined semantics. Compare this with the translation process that human developers perform (where use-cases are translated into implementation by programmers). For robotics, MDD offers far more benefits. We certainly want to be sure the software is correct before we deploy expensive hardware in some environment and risk the physical integrity of the autonomous robot or others in the environment. We need software that is less sensitive to changes in requirements, and model-driven development provides this. We want to enable domain experts (environment experts) to participate more closely in defining the behaviour of autonomous robots. MDD empowers such domain experts as behavioral requirements can be captured in the models (in some cases, the models provide up-to-date documentation). Moreover, MDD provides platform independence and focus on behaviour issues instead of technological details. But then, simulation, validation, and formal verification of the model become far more important than with traditional development (e.g., in a waterfall cycle, the next stage may correct the previous representation of requirements of functionality as it is tested in the implementation).

Modelling the behaviour of a system using state machines has a long history in software development [11]. State machines are central to modelling object-oriented systems since OML [18], they are also used in system engineering languages such as OMG SysML^{TM} [17]. Executable models appeared in executable UML [15]. We have advocated models that specify behaviour through several finite-state machines (FSMs), whose transitions are labeled by predicates of a common-sense logic [4]. We have shown that these models are more succinct that plain FSMs, Petri nets and Behavior Trees [4]. They are easy to comprehend, as the components of FSMs are used in many systems for formal validation or descriptions of automatic devices and protocols. The reactive nature of finite-state machines can be compensated by a reasoning component and domain-knowledge in logic, when we label transitions by a query to an inference engine. Formal verification of behaviour models (model-checking) has traditionally been complicated when several components concurrently operate to define the behaviour of a system. The vector of finite-state machines that constitute such a system can be scheduled deterministically reducing the execution path search space [9]. With more succinct Kripke structures far more model-checking of classic software engineering case studies has been achieved.

Despite this, formal-model checking is costly, and it may not be clear which properties need to be verified. This is where simulation and validation come into play. Simulation allows to identify behaviour that was not considered at the time of requirement elicitation. Trough our simulation, we can discover scenarios where a behaviour specification was incomplete. In this paper we discuss our solution to construct a simulator (and monitor) of a vector of finite-state machines that control autonomous robots (or embedded systems). State machines are core elements in embedded systems and in popular commercial tools widely used in the industry, including *QP*^{TM}[19] *StateWORKS* [24]. While our simulator has characteristics similar of the integration of *MathWorks*^{R} *StateFlow* with *Symlink* we avoid the complex translations that this requires for model-checking [1].

2 Modelling

Designers of robotic system and of autonomous systems naturally model using states and transitions. Thus, a finite-state behaviour model is actually a table.

This transition table consists of 3 columns: (`Source_State`, `Boolean_Expression` `Target_State`). Here, the projection of the transition table on each state is a list, i.e., transitions are considered in (deterministic) sequence. Also, each state of the FSM has 3 sections: **OnEntry**, **OnExit**, and an **Internal** section. One state is designated as the initial state. When a machine is running, it will execute a ringlet. A ringlet is the loop that starts by taking a snapshot (a copy) of all variables. Then the **OnEntry** section is executed (only for the first iteration in a new state, otherwise **OnEntry** is omitted), proceeding to evaluating the Boolean expressions in sequence. If the expression of a transition evaluates to `true`, the transition fires, meaning, the **OnExit** activity is performed and then the state is updated to the target state of the transition, concluding the ringlet in this case. However, if the list of Boolean expression is exhausted and none evaluates to `true`, then the **Internal** section is executed and the ringlet terminates.

Multiple FSMs are executed sequentially in round-robin fashion, in the order they appear in. The interpreter performs one ringlet of one machine before moving to the next one in sequence.

Our interpreter is extremely efficient, capable of performing several thousand transitions per second on-board of an Aldebaran Nao. This particular aspect makes it suitable for evaluating models that not only represent high level behaviours (e.g. the Game Controller of the SPL), but also fast control loops, e.g. the head tracking a ball in robotic soccer.

Our FSMs use simple `C` statements[1] for states, and expressions for transitions (including function calls and evoking expert system evaluation of non-monotonic logic). The communication medium between modules of the software is based on a whiteboard architecture and details have been provided elsewhere [3].

It is important to emphasise that our modelling tool enables three types of variables: local variables within one FSM and variables that are shared by the FSMs in the sequence. The third type are external variables whose value can be updated or modified outside the sequence of machines, but shared with FSMs (e.g. to communicate with sensors or actuators).

3 Illustration

We now present three ubiquitous embedded systems case studies. These have been widely discussed in the literature of software engineering and in particular formal methods. However, we show that simulation and validation discovers emergent behaviour that was not anticipated or constitutes an ambiguity in the specification of requirements. Simulation not only validates the models, but also helps determine the set of properties for formal verification. The first case study will be the *Mine Pump*. We show here that simulation enables to disambiguate the need for 3-position switches, at least for the supervisor. The second case is the *Microwave*, and again, we show that simulation resolves the meaning of the requirement "opening the door stops the cooking." The third case is the

[1] We use the grammar `SimpleC.g` from ANTLR (www.antlr.org).

Industrial Press. We show here that simulation is essential to ensure that the behaviour would be as expected. For these cases, formal model-checking has been performed in the literature, but not all properties were elicited, and thus, models were, in fact, inaccurate (which we discovered through simulation). We focus on model development and simulation (model-checking of these models is discussed elsewhere [9].) Moreover, we highlight pointers to videos of these models running directly on actual robotic platforms (in some cases the same model is being interpreted in two distinct platforms) as promised by model-driven development.

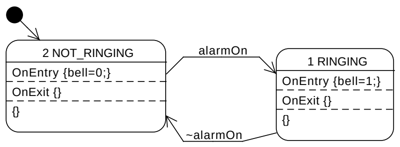
3.1 The Mine Pump

The mining pump is a case widely discussed in the literature [21,22,10,27] where software is controlling a safety-critical system. We follow Burns and Lister [6] where significant formality is provided. Here, we skim over the development of the model [9]. The requirements in natural language [6] are reproduced in Table 1. This study is again illustrative of the power of using FSMs with transitions labeled by a common-sense logic such as DPL. The pump is in one of two states, *running* or *not running*, while simultaneously the alarm is in a *ringing* or *not ringing* state. This is illustrated by the diagrams in Fig 1 (next page).

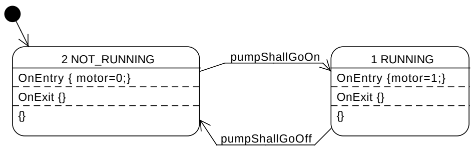
Table 1: Mine Shaft Pump Requirements.

Req.	Description
R1	The pump extracts water from a mine shaft. When the water volume has been reduced below the low-water sensor, the pump is switched off. When the water raises above the high-water sensor it shall switch on.
R2	An human operator can switch the pump on and off provided the water level is between the high-water sensor and the low-water sensor.
R3	Another button accessed by a supervisor can switch the pump on and off independently of the water level.
R4	The pump will not turn on if the methane sensor detects a high reading.
R5	There are two other sensors, a carbon monoxide sensor and an air-flow sensor, and if carbon monoxide is high or air-flow is low, an alarm rings to indicate evacuation of the shaft.

Our techniques reveal several issues with the specification. First, one must inspect the pre-conditions and post-conditions [6], to confirm that the conditions that turn the pump on are not the negation of those that turn it off. In particular, the pump goes on when the water level is high, but remains on when the level drops (Requirement `R1`) until it is below the low sensor before stopping. But it does not re-start as soon as the water level is above the below sensor. Unfortunately, Requirement `R3` is seriously more ambiguous and the language used suggests that the operator's interface as well as the supervisor's interface for controlling the pump are both switches "of the same class" [6] (an assumption also shared by the Behavior Tree approach [10,27]). We argue this first interpretation is inconsistent with the requirements and that simulation reveals that switches that are either on or off, holding only two exclusive states, make no sense. Under this first interpretation (2-state switches), it is possible to construct a model and simulate it. We label the transition from `NOT_RUNNING` to `RUNNING` by the predicate `pumpShallGoOn` which we consider a request to an expert to indicate to us whether the pump shall be running on or not. The expert makes its judgment based on information about the low-water sensor, the high-water sensor, the operator and supervisor buttons, and the methane sensor. In DPL,



(a) States for the alarm.



(b) States for the pump.

Fig. 1: Two FSMs control the mine pump.

having information on all of these inputs is not necessary (but desirable). We state this with the `input` section of the DPL code in Fig. 2.

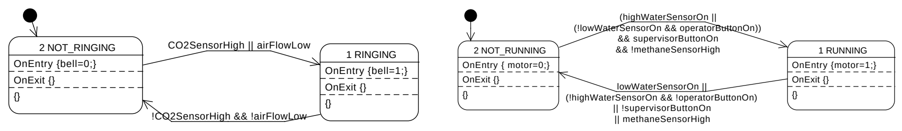
```
name{MINEPUMP}.  input{lowWaterSensorOn}.   input{highWaterSensorOn}.  input{operatorButtonOn}.  input{supervisorButtonOn}.
input{methaneSensorHigh}.

N0: {} =>  ~pumpShallGoOff.                                N1: lowWaterSensorOn =>  pumpShallGoOff.  N1>N0.
N2: {~lowWaterSensorOn,~highWaterSensorOn,~operatorButtonOn}=> pumpShallGoOff.                      N2>N0.
N3: ~supervisorButtonOn => pumpShallGoOff.                 N4: methaneSensorHigh => pumpShallGoOff.  N3>N0.  N4>N0.

P0: {} =>  ~pumpShallGoOn.                                 P1: highWaterSensorOn =>  pumpShallGoOn.   P1>P0.
P2: {~lowWaterSensorOn,~highWaterSensorOn,operatorButtonOn}=> pumpShallGoOn.                         P2>P0.
P3: supervisorButtonOn => pumpShallGoOn.                   P4: ~supervisorButtonOn => ~pumpShallGoOn. P3>P0. P4>P3.
P5: methaneSensorHigh => ~pumpShallGoOn.                                                             P5>P3. P5>P2. P5>P1.

output{b pumpShallGoOn,"pumpShallGoOn"}. output{b pumpShallGoOff,"pumpShallGoOff"}.
```

Fig. 2: DPL coding of the conditions for switching to the state `RUNNING` or to the state of `NOT_RUNNING`.

When shall the pump move from on to off? If we have no information, it shall remain on and not move (Rule `N0`). But if the low-water sensor in on, the pump switches off (this is Rule `N1`, which takes over Rule `N0`). Rule `N2` says that at a water level above *low* and below *high*, the operator can turn the pump off (`N2` takes precedence over `N0`). And the supervisor can turn the pump off, by Rule `N3` $>$ `N0`. A high methane reading turns the pump off as well (Rule `N4`).

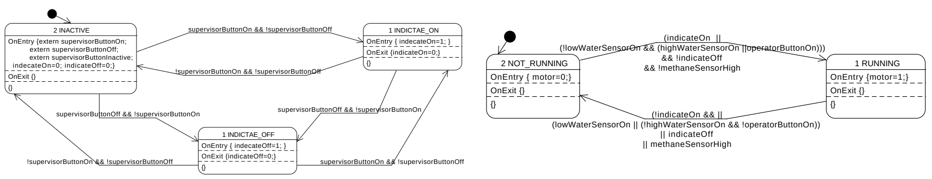
How to answer `pumpShallGoOn`? By default, the pump shall not go on; if asked this question when the pump is off, and in the absence of information, we should not recommend a change of state (this is Rule `P0`). Rule `P1` indicates that if the high-water sensor is on, then the pump goes on and this takes precedence over Rule `P0`. Rule `P2` indicates that the operator can turn the pump on if the water is between levels. Thus, `P2` overwrites `P0`. However, with `P3` and `P4` the supervisor can turn the pump on and off. Nevertheless, all these previous conditions are ruled out if methane is high (Rule `P5`). The reader may have noticed that we have stated Rule `P3` in contradiction with Requirement `R3`. This is because if we add the precedence `P3>P0` and write `P3:supervisorButtonOn=>pumpShallGoOn` enabling the switch of the supervisor to overrule the low-water sensor, compiling these rules obtains `pumpShallGoOff` $\equiv$ `!supervisorButtonOn || methaneSensorHigh`. That is, the water-levels do not matter at all in deciding if the pump shall be off (and therefore the operator is redundant as well). All depends on the methane level which, when high, overrules everything to switch the pump off (or on the supervisor switch that overrules everything else). Similarly, if the supervisor button overrules the operator button, the latter becomes redundant.

So here, one must accept that the low-water sensor takes precedence over the supervisor's switch, which makes sense if running the pump without water is dangerous. The logic theory as written in Fig. 2 provides this functionality. A fact that is verified with simulation in our methodology.



(a) Alarm state transitions. (b) Pump state transitions.

Fig. 3: Two FSMs controlling the mine pump with two state-switches.


2 INACTIVE
OnEntry {extern supervisorButtonOn;
extern supervisorButtonOff;
extern supervisorButtonInactive;
indecateOn=0; indicateOff=0;}
OnExit {}
{}
supervisorButtonOn && !supervisorButtonOff
1 INDICTAE_ON
OnEntry { indecateOn=1; }
OnExit {indicateOn=0;}
{}
!supervisorButtonOn && !supervisorButtonOff
supervisorButtonOff && !supervisorButtonOn
supervisorButtonOff && !supervisorButtonOn
1 INDICTAE_OFF
OnEntry { indecateOff=1; }
OnExit {indicateOff=0;}
{}
!supervisorButtonOn && !supervisorButtonOff
supervisorButtonOn && !supervisorButtonOff
2 NOT_RUNNING
OnEntry { motor=0;}
OnExit {}
{}
(indicateOn ||
(!lowWaterSensorOn && (highWaterSensorOn ||operatorButtonOn)))
&& !indicateOff
&& !methaneSensorHigh
1 RUNNING
OnEntry {motor=1;}
OnExit {}
{}
(!indicateOn && ||
(lowWaterSensorOn || (!highWaterSensorOn && !operatorButtonOn))
|| indicateOff
|| methaneSensorHigh

(a) 3-state supervisor control (3SSC). (b) Receiving signals from a 3SSC.

Fig. 4: The model for the pump under 3-state supervisor control consists of these 2 FSMs and the FSM in Fig. 3a for the alarm.

Compiling this DPL rule system using the `+c` option, we obtain equivalent `C` expressions for when the pump shall be on and for when it shall go off. Namely,

$$\begin{aligned}\texttt{pumpShallGoOff} \equiv\ & \texttt{lowWaterSensorOn} \quad \texttt{||} \quad \texttt{!supervisorButtonOn} \\ & \texttt{||}\ \texttt{methaneSensorHigh} \quad \texttt{||} \quad (\texttt{!highWaterSensorOn}\ \texttt{\&\&}\ \texttt{!operatorButtonOn})\end{aligned}$$

and
$$\begin{aligned}\texttt{pumpShallGoOn} \equiv\ & \texttt{!methaneSensorHigh} \quad \texttt{\&\&} \quad \texttt{supervisorButtonOn} \\ & \texttt{\&\&} \quad (\texttt{highWaterSensorOn} \quad \texttt{||} \quad \{\texttt{!lowWaterSensorOn}\ \texttt{\&\&}\ \texttt{operatorButtonOn}\}).\end{aligned}$$

Notice the asymmetry between the supervisor and operator conditions for pump start! Combining this with the FSM for the alarm results in the model in Fig. 3. Nevertheless, we suggest that there is a second interpretation, i.e. the supervisor's interface is a three state control, that has an inactive state. In the inactive state, it does not overrule any of the other conditions that determine the running of the pump. But the supervisor can activate the switch to on, or off, switching the pump on and off regardless of water levels. That model for supervisor control is presented in Fig. 4 and the new logic theory appears in Fig. 5. Compiling the logic theory gives corresponding equivalent expressions:

$$\texttt{pumpShallGoOn} \equiv \texttt{!methaneSensorHigh \&\&!indicateOff\&\&} \left(\texttt{indicateOn.||} \left\{\texttt{!lowWaterSensorOn\&\&} \left[\texttt{highWaterSensorOn|| operatorButtonOn}\right]\right\}\right).$$

```
pumpShallGoOff  ≡ methaneSensorHigh   ||    indicateOff   || (!indicateOn
                  && {lowWaterSensorOn||    (!highWaterSensorOn   && ! operatorButtonOn)} .
```

Replacing the transition labels in Fig. 1b, we obtain our model for the second interpretation of this case study. The Behavior Tree approach fails both interpretations of R3 and did not even verify any property regarding R3 [10,27].

Simulation uncovers these observations during development and helps define the properties to use for model-checking. This contrasts the properties under the two interpretations for the switches [9]. Finally, we can validate our models on actual platforms. We have enabled our interpreter to control a Lego NXT and built a model of the pump with touch-sensors and floating balls that, when the water level rises high enough, will trigger the sensor. Unfortunately the NXT only has 4 ports for sensors, thus we do not observe the behaviour of the alarm. Also, the 3-state supervisor button is simulated by cycling trough indicating on, off, and inactive for each momentary push of the corresponding touch sensor. Although this is a rather improvised artefact with unreliable sensors, we have a video that shows that the behaviour of the pump under the control of the model in Fig. 4 is correct. We trust the correctness derived from model checking as the utmost proof. A video of a pump executing the model appears on youtu.be/y4muLP0jA8U. We believe claims in the literature regarding the correctness of the model were incomplete because there was no simulation of the model that would have uncovered the above requirement issues.

```
name{MINEPUMP}.
input{lowWaterSensorOn}.  input{highWaterSensorOn}.  input{operatorButtonOn}.  input{methaneSensorHigh}.  input{indicateOn}.
input{indicateOff}.
P0: {} =>  ~pumpShallGoOn.            P1: highWaterSensorOn =>  pumpShallGoOn.                 P1>P0.
P2: lowWaterSensorOn =>  ~pumpShallGoOn.                                                       P2>P1.
P3: {~lowWaterSensorOn,~highWaterSensorOn,operatorButtonOn}=> pumpShallGoOn.                   P3>P2. P3>P0.
P4: {~lowWaterSensorOn,~highWaterSensorOn,~operatorButtonOn}=> ~pumpShallGoOn.                 P4>P3.
P5: indicateOn => pumpShallGoOn.      P6: indicateOff => ~pumpShallGoOn.                       P5>P2. P5>P4. P5>P0. P6>P5.
P7: methaneSensorHigh => ~pumpShallGoOn.                                                       P7>P5. P7>P3. P7>P1.
N0: {} =>  ~pumpShallGoOff.           N1: {~indicateOn,lowWaterSensorOn} =>  pumpShallGoOff.   N1>N0.
N2: {~indicateOn,~lowWaterSensorOn,~highWaterSensorOn,~operatorButtonOn}=> pumpShallGoOff.     N2>N0.
N3: indicateOff => pumpShallGoOff.    N4: methaneSensorHigh => pumpShallGoOff.                 N3>N0.  N4>N0.
output{b pumpShallGoOn,"pumpShallGoOn"}. output{b pumpShallGoOff,"pumpShallGoOff"}.
```

Fig. 5: DPL coding of the conditions for switching to the state RUNNING using a 3-state supervisor control.

3.2 The Microwave

A microwave is an example in software engineering [23] to illustrate modelling through states and transitions, but is also present in the context of model-checking [7, Page 39] as the safety feature of *disabling radiation when the door is open* is an analogous requirement to the case of faulty software on the Therac-25 radiation machine that caused harm to patients [2, Page 2]. The embedded software for the behaviour of a microwave oven is not only widely discussed in software modelling [14,20,24] but also as part of behaviour engineering [25,8,16].

Details of this example have been discussed previously [5]. Suffice it to say that one of the requirements is that opening the door shall stop the cooking. But this says nothing about what happens with the timer, and through modelling and simulation we discover that opening the door could *pause* the timer, or could *clear* the timer, or could *not affect* the timer at all, which continues to count down. Modelling by FSMs, where the labels for transitions can be statements in a logic that demand proof, has been contrasted with plain FSMs, Petri nets, and Behavior Trees (relevant behaviour modelling techniques in software engineering) using this very prominent example [4]. FSMs produce smaller models, and clarify requirements. This is because the ambiguity of the requirements is detected at the simulation of the model. The model-driven approach enables the automatic generation of implementations for diverse platforms and programming languages, e.g., the same models can generate code in Java for a Lego Mindstorm (`youtu.be/iEkCHqSfMco`) as well as C++ for a Nao (`youtu.be/Dm3SP3q9_VE`). Note, we can use a robot to simulate the behaviour of a microwave in the same way as our visual simulation tool.

3.3 The Industrial Press

The industrial metal press [13] has been studied in the literature of model checking for failure analysis [10,12,26]. Table 2 reproduces the requirements [26]. Once

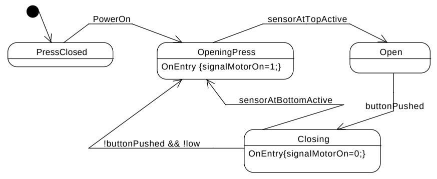
Table 2: Industrial Metal Press Requirements.

Req.	Description
R1	The plunger is initially resting at the bottom with the motor off.
R2	When power is supplied, the controller shall turn the motor on, causing the plunger to rise.
R3	When at the top, the plunger shall be held there until the operator pushes and holds down the button. This shall cause the controller to turn the motor off and the plunger will begin to fall.
R4	If the operator releases the button while the plunger is falling slowly (above PONR), the controller shall turn the motor on again, causing the plunger to start rising again, without reaching the bottom.
R5	If the plunger is falling fast (below PONR) then the controller shall leave the motor off until the plunger reaches the bottom.
R6	When the plunger is at the bottom the controller shall turn the motor on: the plunger will rise again.

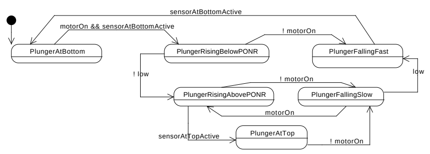
again, later papers [10,26] that cite the original source [13] capture requirements differently, or incompletely. For example, the requirement that once the plunger has come down, it shall stay down until the "human operator releases the button and inserts and removes sheets" [12]. In the original source [13] there is even an additional infrared-line sensor; once the plunger is down, the button must be released by the operator and the line cut for the plunger to move up again.

Because the On/Off button is not modelled correctly, and the infrared-link is not modelled at all, modelling as described in the Design Behavior Tree (DBT) [26, Figure 4 and HighResolution gif] results in a pathological cyclical behaviour of the system: while the human operator keeps the button pushed (and the power is on), the plunger rises to the top (with the motor on), reaches the top (the motor goes off) falls down, and rises up again.

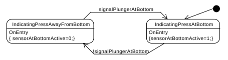
First, we have constructed a model that reproduces the DBT [26, Fig. 4] to focus on the model checking aspects. This model that mimics such a DBT appears in Fig. 6. We will not elaborate here on the formal verification of the model. We rather emphasise that the properties verified in the literature are not sufficient to completely validate the model. However, our analysis with a simulator does suggest a correction (shown in Fig. 7) that enables to remove the anomaly. See youtu.be/FpVUSrvLI0c for a video of the simulator operating concurrently on all the FSMs of the model for the corrected version. The video shows the progress of the FSMs, while sensor data is supplied trough the monitoring tool. The host computer acts as a robot through our whiteboard architecture and produces speech also from the FSMs. The plunger raises to the top and return back when falling slowly when the operator releases the button. However, once below the PONR, the release of the operator does not affect the fall. This again shows that our approach compares favourably with Behavior Trees. To further illustrate the effectiveness of modelling and simulation, we have deployed these models for execution on two platforms, a Nao robot and a NXT. A video (`youtu.be/blUpMdH14pM`) of the system emulating the behaviour as formalised from the Behavior Tree approach and its corrections is available. We emphasise that simulation is not a replacement for formal verification but it does enable elicitation of properties for a later stage of formal verification. Such model-checking is described elsewhere [9].



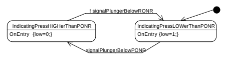
(a) States for the Controller.



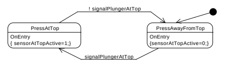
(b) States for the Plunger.


signalPlungerAtBottom
IndicatingPressAwayFromBottom
OnEntry
{ sensorAtBottomActive=0;}
IndicatingPressAtBottom
OnEntry
{sensorAtBottomActive=1;}
!signalPlungerAtBottom

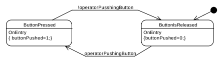
(c) Bottom Sensor.



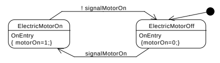
(d) PONR Sensor.



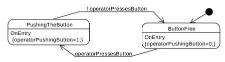
(e) Top Sensor.



(f) Button.



(g) Electric Motor.



(h) States of the Operator.

Fig. 6: Complete model of the industrial press that mimics the Design Behavior Tree [26, Fig. 4 and high-res gif] .

4 Simulation and Monitoring Tools

The execution of the models is performed by an interpreter based on data structures that represent a vector of FSMs in standard `C++`. The data structures for each FSM represent the states and the transitions, as well as the ANTLR expression tree for transition labelling. This is not only useful to obtain the Kripke structure of a vector of FSMs and perform model-checking, but also as an interpreter used for simulation. We have extended the publicly available `qfsm`

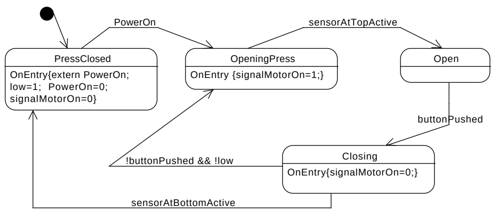
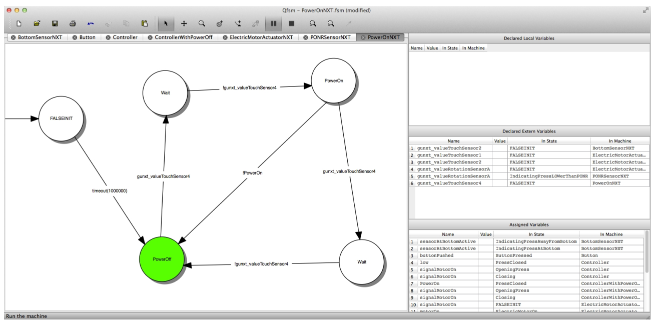


Fig. 7: Corrected model from Fig. 6a (based on DBT modelling from the literature). This model correctly pauses for a signal that restarts the system.

(qfsm.sf.net) source code to implement the semantics of our FSMs. First, the states have the **OnEntry**, **OnExit** and **Internal** sections and the transitions can be Boolean expressions. However, the most important direction is that, while this tool was originally simply an editor of FSMs, we have converted it into a visual simulator of finite state machines.

Our tool enables the visualisation of the execution of a vector of FSMs highlighting the running state while keeping the interface responsive (this required additional Qt signals and slots). Since the model of a system is a vector of FSMs, we created a tabbed interface to enable the rapid visualisation for each FSM involved as a component of a larger model.

The simulator has a tightly coupled monitor that displays the values of the variables used in the FSMs (locals, globals shared, and external ones on the whiteboard. This enables the visualisation of the effects on the values on the variables as the simulation progresses. The update of the variables is performed asynchronously by calling code to update the variable table when the syntax checker has finished parsing a state. The integration of the interpreter demanded the sub-classing of some interpreter classes and since ANTLR generates `C` code, a `C++` wrapper was written. A video (youtu.be/rceNij4IkJE) demonstrates usage of the simulator and how a vector of FSMs is constructed, the syntax of the simple `C` language are verified and the simulator operates.



Declared Local Variables

Name	Value	In State	In Machine

Declared Extern Variables

	Name	Value	In State	In Machine
1	gunxt_valueTouchSensor2		FALSEINIT	BottomSensorNXT
2	gunxt_valueTouchSensor1		FALSEINIT	ElectricMotorActua...
3	gunxt_valueTouchSensor2		FALSEINIT	ElectricMotorActua...
4	gunxt_valueRotationSensorA		FALSEINIT	ElectricMotorActua...
5	gunxt_valueRotationSensorA		IndicatingPressLOWerThanPONR	PONRSensorNXT
6	gunxt_valueTouchSensor4		FALSEINIT	PowerOnNXT

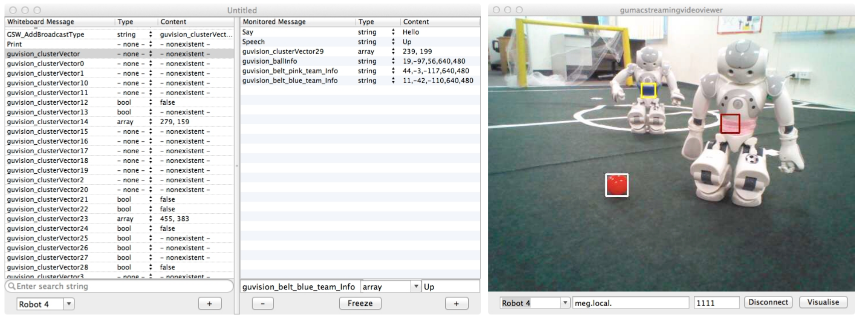
Assigned Variables

	Name	Value	In State	In Machine
1	sensorAtBottomActive		IndicatingPressAwayFromBottom	BottomSensorNXT
2	sensorAtBottomActive		IndicatingPressAtBottom	BottomSensorNXT
3	buttonPushed		ButtonPressed	Button
4	low		PressClosed	Controller
5	signalMotorOn		OpeningPress	Controller
6	signalMotorOn		Closing	Controller
7	PowerOn		PressClosed	ControllerWithPowerO...
8	signalMotorOn		OpeningPress	ControllerWithPowerO...
9	signalMotorOn		Closing	ControllerWithPowerO...
10	signalMotorOn		FALSEINIT	ElectricMotorActuato...
11	motorOn		ElectricMotorOn	ElectricMotorActuato...

Fig. 8: The interface of the simulator with the vector of FSMs for the an *Ambulatory Insulin Pump* and with the tab for the *Volume Sensor* active.

The complimentary aspect to the simulator is a remote monitor that enables to visualise the execution of a vector of FSMs on an actual robot (Fig 9).

This requires a distributed whiteboard that we have implemented using a time-triggered architecture and whose description will be presented elsewhere. With this idea, simulation is possible while tracing or monitoring the state of the execution, especially the values of the many variables involved in a complex model of several FSMs running in real-time. This monitor inspects the state of simulation by inspecting the message passing that happens with variables on our distributed whiteboard architecture. With this distributed whiteboard technology, is also possible to visualise in a model (a vector of FSMS) the switching states from our extended `qfsm`, although the vector of FSMs is executing remotely on a robotic platform. One important aspect of our monitor tool and



Whiteboard Message	Type	Content
GSW_AddBroadcastType	string	guvision_clusterVect...
Print	– none –	– nonexistent –
guvision_clusterVector	– none –	– nonexistent –
guvision_clusterVector0	– none –	– nonexistent –
guvision_clusterVector1	– none –	– nonexistent –
guvision_clusterVector10	– none –	– nonexistent –
guvision_clusterVector11	– none –	– nonexistent –
guvision_clusterVector12	bool	false
guvision_clusterVector13	bool	– nonexistent –
guvision_clusterVector14	array	279, 159
guvision_clusterVector15	– none –	– nonexistent –
guvision_clusterVector16	– none –	– nonexistent –
guvision_clusterVector17	– none –	– nonexistent –
guvision_clusterVector18	– none –	– nonexistent –
guvision_clusterVector19	– none –	– nonexistent –
guvision_clusterVector2	– none –	– nonexistent –
guvision_clusterVector20	– none –	– nonexistent –
guvision_clusterVector21	bool	false
guvision_clusterVector22	bool	false
guvision_clusterVector23	array	455, 383
guvision_clusterVector24	bool	false
guvision_clusterVector25	bool	– nonexistent –
guvision_clusterVector26	bool	– nonexistent –
guvision_clusterVector27	bool	– nonexistent –
guvision_clusterVector28	bool	false
guvision_clusterVector3	– none –	– nonexistent –

Monitored Message	Type	Content
Say	string	Hello
Speech	string	Up
guvision_clusterVector29	array	239, 199
guvision_ballInfo	string	19,-97,56,640,480
guvision_belt_pink_team_Info	string	44,-3,-117,640,480
guvision_belt_blue_team_Info	string	11,-42,-110,640,480
guvision_belt_blue_team_Info	array	Up

Fig. 9: Monitoring of variables with distributed whiteboard and monitoring behaviour in the robot (remote monitoring of vision analysis inside a Nao robot that is looking at other robots (ball and team colour analysis).

our simulation tool is that the monitoring and the execution of the behaviour (a vector of FSMs) are the same, whether this is a simulation outside the robot or actual execution on the intended platform. The closest analogue to our ideas is Aldebaran's Coreographe. However, first, this is platform specific: the simulator is effective only for specific Nao robot versions. Aldebaran has chosen `SOAP` incurring large penalties to the relay of communication of its monitoring software. Because its wide open semantics, formal verification would result in exponential state explosion and is thus infeasible for formal model-checking approaches.

5 Conclusions

We have shown that simulation is a highly desirable tool to complement formal verification for system behaviour validation. We believe that earlier work on verification of correctness of models for the case studies discussed here has been incomplete because a faithful simulator has not been used. As a result, some properties are verified formally, but some emergent behaviour passes undetected (possibly until deployment).

References

1. A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electr. Notes Theor. Comput. Sci.*, 109:43–56, 2004.

2. C. Baier and J.-P. Katoen. *Principles of model checking.* MIT Press, 2008.
3. D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock. Architecture for hybrid robotic behavior. In *4th Int. Conf. on Hybrid Artificial Intelligence Systems HAIS*, LNCS 5572, p. 145–156. Springer-Verlag, 2009.
4. D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock. Non-monotonic reasoning for requirements engineering. In *Proc. 5th Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE)*, p. 68–77, Athens, 2010. SciTePress.
5. D. Billington, V. Estivill-Castro, R. Hexel, and R. Rock. Modelling behaviour requirements for automatic interpretation, simulation and deployment. In *SIMPAR 2nd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots*, LNCS 6472 p. 204–216. Springer, 2010.
6. A. Burns and A.M. Lister. A framework for building dependable systems. *The Computer Journal*, 34(2):173–181, 1991.
7. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press, 2001.
8. R. G. Dromey and D. Powell. Early requirements defect detection. *TickIT Journal*, 4Q05:3–13, 2005.
9. V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth. Efficient model checkign and fmea analysis with deterministic scheduling of transition-labeled finite-state machines. *3rd World Congress Software Engineering*, China, 2012. to appear.
10. L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay. Experience with fault injection experiments for FMEA. *Software, Practice and Experience*, 41(11):1233–1258, 2011.
11. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.
12. T. Mahmood and E. Kazmierczak. A knowledge-based approach for safety analysis using system interactions. *13th Asia Pacific Software Engineering Conf., APSEC 2006.* , p. 445 –452.
13. T. McDermid, J.and Kelly. Industrial press: Safety case. Technical report, High Integrity Systems Engineering Group, University of York, 1996.
14. S. J. Mellor. Embedded systems in UML. OMG White paper, 2007. www.omg.org/news/whitepapers/ label: We can generate Systems Today.
15. S. J. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture.* Addison-Wesley Publishing Co., Reading, MA, 2002.
16. T. Myers and R. G. Dromey. From requirements to embedded software - formalising the key steps. In *20th Australian Software Engineering Conf. (ASWEC)*, p. 23–33, Gold Cost, Australia, 2009. IEEE Computer Society.
17. OMG. OMG systems modeling language (OMG SysML^{TM}). Version 1.3 with change bars, June 2012.
18. J. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modelling and Design.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
19. M. Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems.* Newnes, 2008.
20. S. Shlaer and S. J. Mellor. *Object lifecycles : modeling the world in states.* Yourdon Press, Englewood Cliffs, N.J., 1992.
21. S.K. Shrivastava, Mancini L. V., and B. Randell. The duality of fault-tolerant system structures. *Software — Practice and Experience*, 23(7):773–798, 1993.
22. M. Sloman and J. Kramer. *Distributed systems and computer networks.* Prentice Hall (UK), 1987.
23. I. Sommerville. *Software engineering (9th ed.).* Addison-Wesley, Boston, 2010.
24. F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach.* CRC Press, NY, 2006.

25. L. Wen and R. G. Dromey. From requirements change to design change: A formal path. In *2nd Int. Conf. on Software Engineering and Formal Methods (SEFM 2004)*, p. 104–113, Beijing, IEEE Computer Society.
26. K. Winter and N. Yatapanage. The metal press case study. Technical report, University of Queensland. supplement in `www.itee.uq.edu.au/~docs/FMEA`.
27. K. Winter and N. Yatapanage. The mine pump case study. Technical report, University of Queensland. supplement in `www.itee.uq.edu.au/~docs/FMEA`.